Estimating Municipalities' GDP Using Machine Learning

Patrick Alves[1]

João de Negri[2]

We estimate the Counties' GDP in Brazil using ten different machine learning algorithms and hyperparameter tunning. This allowed us to compare the performance of Random Search and Grid Search methods for optimal hyperparameter tuning. We find that the hyperparameter optimization using Random Search allowed very satisfactory results. For both tuning methods, the Extreme Learning Machine (ELM), k-Nearest Neighbors (KNN) and Multilayer Perceptron (MLP) stood out as the most accurate ones, although the training time in Grid Search is significantly higher.

Keywords: hyperparameter tuning for regression; municipalities' GDP.

1. Introduction

In a continent-sized country like Brazil, the availability of gross domestic product disaggregated at the city level is an important tool, both for local authorities and business entrepreneurs, seeking to make decisions in line with regional economic growth. In Brazil, due to its size, the federal government (Union) must rely on the local structure provided by 5.670 municipalities to implement many of its public policies. Also, the municipalities are heavily dependent on resources from the federal government, sometimes as the only source to maintain local security, health, and educational infrastructure. The resource transfers from the Union to municipalities are conditioned to fiscal accountability, measured as ratios normalized to municipalities' GDP. As such, providing accurate and updated estimates of municipalities' GDP is of ultimate importance in Brazil.

The last official disclosure of municipalities' GDP was in 2017. We aim to update this for the years 2018 and 2019 using the power of the machine learning (ML) algorithms and hyperparameter tuning.

Hyperparameter tuning has become one of the main challenges in data science practice. The hyperparameter is not estimated during the learning process. Without hyperparameter tuning, one must run the same algorithm multiple times and track the best accuracy statistics afterward. Since this is a very time-consuming process, previous research can help to restrict the parameter's search space, speeding up the time for hyperparameter tuning.

[1] Researcher at Brazilian Institute of Applied Economic Research

[2] Researcher at Brazilian Institute of Applied Economic Research

Hyperparameters optimization is also an important task within Automated Machine Learning (AutoML) (Feurer & Hutter, 2019.). Additional research in this field can contribute to reducing human efforts necessary for applying ML. As such, the contribution of this paper is twofold. We compare the performance of Random Search and Grid Search methods for optimal hyperparameter tuning using ten different machine learning algorithms and we provide the updated estimates of municipalities' GDP for more recent years.

2. Database

The database used to train the algorithms has 44.560 observations, being 5.570 municipalities in 8 years. As for now, 2017 is the last available year, and it is published by the Brazilian Institute of Geography and Statistics with 4 years delay. As the target variable, we choose the natural logarithm of the GDP in 2017. As the features, we choose the natural logarithm of six GDP lags and the quadratic transformation of those natural logarithms. The updated database for the years 2018 and 2019 will be kept available in the repository: https://github.com/estatistica/dados.

3. Hyperparameters tunning

Some recent advances to the state of the art in ML have come from better configurations of existing techniques rather than novel approaches (Bergstra et. Al., 2011). A very sophisticated ML algorithm with non-optimal hyperparameters can perform worse than a primitive algorithm with optimal hyperparameters. Unlike the model parameters, the hyperparameters are not estimated during the learning process and must be chosen beforehand. As such the strategy for hyperparameters optimization is of ultimate importance in achieving the best performance.

The Bayesian Search (BS), Grid Search (GS), and Random Search (RS) are some of the current popular strategies for hyperparameters optimization. Grid search exhaustively re-estimates the ML on every possible combination of the grid space, stores the accuracy metrics, and then chooses the hyperparameters associated with the best accuracy.

Instead of trying every possible combination in the parameter space, Random Search (RS) randomly draws the hyperparameters from the parameters space. This greatly improves the computational time required to find the optimal hyperparameters, but at the coast.

In Bayesian tunning, the result from previous runs helps to improve the next experiment. It relies on a previously defined objective function to interactively find the optimal hyperparameters. At each interaction, the objective function is updated using the hyperparameters from the previous step (Brochu, Cora, and Freitas, 2010). Even though Grid Search is an exhaustive method, recent literature has pointed that the chances of finding the optimal parameter are higher in Random Search (RS). Random search also works well for lower dimensional data, and when for a relatively smaller number of dimensions (Bergstra and Bengio, 2012).

The current applications of Bayesian Search in ML usually focus on the tunning of only one hyperparameter. For instance, in Multi-task Elastic-Net we would optimize only the alpha parameter while keeping the other parameters fixed. Bayesian Search is also time-consuming and will not be cover in this paper.

4. Random Search versus Grid Search

The effects of hyperparameters may depend on the dataset size, the number of features, target variable (binary, continuous, multicategory), and the algorithm itself. In practice, one usually tries a different combination of many hyperparameters at once. It is never clear what is the impact of each parameter, and what is the individual contribution in avoiding overfitting the data. An intensive explanation about the purpose of each parameter in each of the ML algorithms is beyond the scope of this paper. Instead, based on our dataset size, we will define a generous parameter search space and compare the performance and execution time for the Grid Search and Random Search methods available in scikit-learn library in python.

Table 1 shows the space parameters choice for each of the ML algorithms. For the exercise to make sense, we had to restrict hyperparameter space when using the Grid Search method, otherwise, the exhaustive hyperparameter search would look improper in everyday practice. When one chooses the Grid Search, the focus turns to the regularization parameters and activation functions. We try to emulate this behavior by letting some parameters receive their default values in scikit-learn.

As for the tolerance and regularization parameters, we use the same space amplitude as Random Search, but with a higher distance between the values. For instance, the alpha parameter in Multi-task Elastic-Net goes from 0.005 to 5.705 by an increment of 0.3 in Random Search and by an increment of 0.6 in Grid Search. Additionally, for the Multilayer Perceptron regression (MLP) and Extreme Learning Machine (MLP), we also restricted the number of layers, since the parameter space would burst the python limit without this restriction.

For random search tunning, we choose a sample of size 75 from the hyperparameter space. For the grid search, the parameter space varies between 1500 and 350000. Table 1 shows the hyperparameter space chosen for the two methods.

Table 1: ML Algorithms and Hyperparameters Space.

ML Algorithm	Hyperparameters Search Space	Grid Search Space

Multi-task Elastic-Net (Multi-Task)	alpha = [0.005 to 5.705 by 0.3]
	l1-ratio = [0.033 to 9.7 by 0.33]
	normalize = [True, False]
	fit-intercept= [True, False]
	warm-start= [True, False]
	copy-X= [True, False]
	tolerance = [0.0002 to 0.0034 by 0.0004]
	selection= [cyclic, random]
		alpha = [0.005 to 5.705 by 0.3]
		l1-ratio = [0.033 to 9.7 by 0.33]
		normalize = False
		fit-intercept= True
		warm-start= False
		copy-X= True
		tolerance = [0.0002 to 0.0034 by 0.0008]
		selection= [cyclic, random]

Least-Angle Regression (LARS)	alpha = [0.2 to 40 by 0.2]
	fit-intercept= [True, False]
	fit-path= [True, False]
	normalize = [True, False]
	copy-X= [True, False]
	positive= [True, False]
	eps = [10 to 100 by 0.1]
		alpha = [0.0033 to 3 by 0.013]
		fit-intercept= True
		fit-path= True
		normalize = True
		copy-X= True
		positive= False
		eps = [10 to 100 by 0.1]

Stochastic Gradient Descendent (SGD)	alpha = [0.04 to 100 by 0.04]
	l1-ratio = [0.025 to 0.975 by 0.025]
	loss = [squared loss, huber, epsilon insensitive, squared epsilon insensitive]
	penalty = [l2, l1, elastic-net]
	epsilon = [0.2 to 20 by 0.2]
	learning-rates = [constant, optimal, invscaling, adaptive]
	eta0 = [0.2 to 20 by 0.2]
	power-t = [0.025 to 1 by 0.025]
	early_stopping = [False,True]
		alpha = [0.8 to 10 by 0.4]
		l1-ratio = [0.05 to 0.23 by 0.02]
		loss = squared loss
		penalty = l2
		epsilon = [0.014 to 0.27 by 0.03]
		learning-rates = invscaling
		eta0 = [0.2 to 20 by 0.2]
		power-t = [0.025 to 1 by 0.025]
		early_stopping = [False,True]

least squares with l2 regularization (Ridge)	alpha = [0.04 to 2500 by 0.04]
	fit-intercept = [True, False]
	normalize = [True, False]
	copy-X = [True, False]
	solver= [auto, svd, cholesky, lsqr, sparse_cg]
	tolerance = [1e-07 to 1e-04 by 1e-07]
		alpha = [0.016 to 0.2 by 0.01]
		fit-intercept = [True, False]
		normalize = [True, False]
		copy-X = [True, False]
		solver= [auto, svd, cholesky, lsqr, sparse_cg]
		tolerance = [1e-05 to 0.0099 by 1e-04]

Regression with l1 and l2 regularizer (Elastic-Net)	alpha = [0.04 to 2500 by 0.04]
	l1-ratio = [0.04 to 2500 by 0.04]
	fit-intercept = [True, False]
	normalize = [True, False]
	copy-X = [True, False]
	precompute = [True, False]
	warm-start = [True, False]
	positive= [True, False]
	tolerance = [1e-07 to 1e-04 by 1e-07]
	selection= [cyclic, random]
		alpha = [0.04 to 2500 by 0.04]
		l1-ratio = [0.04 to 2500 by 0.04]
		fit-intercept = [True, False]
		normalize = [True, False]
		copy-X = [True, False]
		precompute = [True, False]
		warm-start = [True, False]
		positive= [True, False]
		tolerance = [1e-07 to 1e-04 by 1e-07]
		selection= [cyclic, random]

Bayesian Ridge (Bayes)	alpha-1 = [0.02 to 20 by 0.02]
	alpha-2 = [0.02 to 20 by 0.02]
	lambda-1 = [0.02 to 20 by 0.02]
	lambda-2 = [0.02 to 20 by 0.02]
		alpha-1 = [0.02 to 20 by 0.02]
		alpha-2 = [0.02 to 20 by 0.02]
		lambda-1 = [0.02 to 20 by 0.02]
		lambda-2 = [0.02 to 20 by 0.02]

	compute-score = [False,True]
	copy-X = [False,True]
	fit-intercept = [False,True]
	normalize= [False,True]
	tolerance = [1e-07 to 1e-02 by 1e-07]
		compute-score = [False,True]
		copy-X = [False,True]
		fit-intercept = [False,True]
		normalize= [False,True]
		tolerance = [1e-07 to 1e-02 by 1e-07]

Least Absolute Shrinkage Selection Operator (Lasso)	alpha = [0.001 to 0.1 by 0.005]
	fit-intercept = [True, False]
	copy-X = [True, False]
	normalize = [True, False]
	precompute = [True, False]
	positive= [True, False]
	selection = [cyclic, random]
	tolerance = [0.0001 to 0.01 by 0,00001]
		alpha = [0.001 to 0.1 by 0.005]
		fit-intercept = True
		copy-X = True
		normalize = False
		precompute = False
		positive= False
		selection = [cyclic, random]
		tolerance = [0.0001 to 0.01 by 0,00002]

Support Vector Machine (SVM)	kernel = [linear, poly, rbf, sigmoid, precomputed]
	degree = [0.02 to 0.492 by 0.01]
	gamma = [scale, auto]
	tolerance = [0.0005 to 0.0955 by 0.005]
	coef0 = [0.02 to 2 by 0.08]
	C = [0.02 to 2 by 0.08]
	shrinking= [True, False]
	epsilon = [0.02 to 2 by 0.02]
		kernel = [linear, poly, rbf, sigmoid, precomputed]
		degree = [0.02 to 0.492 by 0.01]
		gamma = [scale, auto]
		tolerance = [0.0005 to 0.0955 by 0.005]
		coef0 = [0.02 to 2 by 0.08]
		C = [0.02 to 2 by 0.08]
		shrinking= [True, False]
		epsilon = [0.02 to 2 by 0.02]

k-Nearest Neighbor (KNN)	n-neighbors= [4 to 10 by 1]
	weights = [uniform, distance]
	algorithm = [auto, ball-tree, kd-tree, brute]
	leaf-size = [10 to 50 by 1]
	p = [1 to 8 by 1]
		n-neighbors= [4 to 10 by 2]
		weights = uniform
		algorithm = [auto, ball-tree, kd-tree, brute]
		leaf-size = [10 to 50 by 2]
		p = [1 to 8 by 2]

Multi-layer Perceptron (MLP)	hidden-layer-sizes = [(100,50), (200,25), (25,100,25), (50,50,50), (75,50,75), (80,60,80)]
	activation = [identity, logistic, tanh, relu]
	solver = [lbfgs, sgd, adam]
	alpha = [0.001 to 1 by 1]
	learning-rate = [constant, invscaling, adaptive]
	learning-rate-init = [0.0002 to 4.0 by 0.0002]
	power-t = [0.02 to 4.0 by 0.02]
	tolerance = [1e-06 to 9.9e-05 by 1e-06]
	momentum = [0.01 to 0.5 by 0.01]
	nesterovs-momentum= [True,False]
	early-stopping= [True,False]
	warm-start = [True,False]
	beta-*1* = [0.01 to 0.5 by 0.01]
	beta-*2* = [0.01 to 0.5 by 0.01]
	epsilon = [0.01 to 0.5 by 0.01]
		hidden-layer-sizes = [(100,50), (200,25), (25,100,25), (50,50,50), (75,50,75), (80,60,80)]
		activation = [identity, logistic, tanh, relu]
		solver = [lbfgs, sgd, adam]
		alpha = [0.001 to 1 by 1]
		learning-rate = [constant, invscaling, adaptive]
		learning-rate-init = [0.0002 to 4.0 by 0.0002]
		power-t = [0.02 to 4.0 by 0.02]
		tolerance = [1e-06 to 9.9e-05 by 1e-06]
		momentum = [0.01 to 0.5 by 0.01]
		nesterovs-momentum= [True,False]
		early-stopping= [True,False]
		warm-start = [True,False]
		beta-*1* = [0.01 to 0.5 by 0.01]
		beta-*2* = [0.01 to 0.5 by 0.01]
		epsilon = [0.01 to 0.5 by 0.01]

Extreme Learning Machine (ELM)	activation function = [tanh, sine, tribas, sigmoid, hardlim, softlim, gaussian, multiquadric, inv. multiquadric]
	hidden layers = [2 to 20 by 2]
	rbf-width = [0.0125 to 0.95 by 0.0625]
	activation = [{power:2.5}, {power:3.0}, {power:3.5}, {power:4.0}, {power:4.5}]
		activation function = [tanh, sine, tribas, sigmoid, hardlim, softlim, gaussian, multiquadric, inv. multiquadric]
		hidden layers = [2 to 20 by 4]
		rbf-width = [0.0125 to 0.8875 by 0.125]

		activation = [{power:1.5}, {power:2.5}, {power:3.0},{power:3.5},{power:4.0}, {power:4.5},{power:5.5},{power:6.5}]

5. Performance Results

The Explained Variance Score (EVS), R-square (R2), Mean Absolute Error (MAE), accuracies measures presented in this section are given by:

$$EVS = 1 - \frac{VAR(y_i - \hat{y}_i)}{VAR(y_i)}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

$$MAE = 1 - \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

While the MAE measures the average of the residuals in the dataset, the R2 represents the proportion of the variance in the dependent variable which is explained by the model. EVS also measures the proportion of variance explained by the model. But EVS doesn't use the average value ($\bar{y}$), thus not being influenced by extreme values in the dataset.

Table 2 shows the metrics Explained Variance Score (EVS), R-square (R2), Mean Absolute Error (MAE), and the training time for both the tuning methods. The training time is relevant when we compare the same algorithms using different tunning methods or different algorithms using the same tunning methods. On the other hand, the metrics EVS, R2, and MAE are always relevant, independent of the tunning methods.

Table 2: EVS, R^2, MAE and Training Time for Random Search and Grid Search

	Random Search				Grid Search			
Algorithm	EVS	R^2	MAE	Time	EVS	R^2	MAE	Time
MLP	0.9553	0.9552	0.0177	1h19min35s	0.9582	0.9582	0.0172	15h9min36s
Lasso	0.9565	0.9084	0.0355	658ms	0.9540	0.9540	0.0915	4min44s
Multi-Task	0.9509	0.9509	0.0868	2.33s	0.9607	0.9606	0.0802	4min7s
Bayes	0.9591	0.9590	0.0177	1.21s	0.9559	0.9559	0.0847	14h2min14s
Elastic Net	0.9543	0.8855	0.0410	2.92s	0.9552	0.9552	0.0850	16h47min29s
Ridge	0.9533	0.9532	0.0861	2.37s	0.9538	0.9538	0.0192	1min30s
LARS	0.9482	0.9482	0.0903	2.45s	0.1247	0.1247	0.5262	2h9min2s
SGD	0.8968	0.8968	0.1459	9.87s	0.4832	0.4498	0.0876	2h4min41s
SVM	0.8229	0.8422	0.0495	7.7s	0.8181	0.7601	0.0583	3h40min26s
ELM	0.9524	0.9524	0,0864	3.28s	0.9610	0.9610	0.0818	1min53s
KNN	0.9552	0.9551	0.0193	2min35s	0.9510	0.9507	0.0951	22min46s

Source: Elaborate by the authors

As expected, the training time is always higher in the Grid Search, but not the EVS. Even when the EVS is higher in the Grid Sear, the training time is not worth it. For the Random Search method, the algorithms Lasso, Bayes, Elastic-Net, Ridge, and LARS algorithms perform satisfactory well in terms of both R2 and Training Time. The best performance was achieved by Multi-layer Perceptron (MLP), with a training time of 1h 19 min 35s.

Table 3 presents a summary of all algorithms used. The chosen hyperparameters are presented in Appendix A and B.

Table 3: Training Summary for the Algorithms

Algorithm	Summary
MLP	Best accuracy (MAE) for both RS and GS, although at a very high time in GS.
Lasso	The smaller training time in RS.
Multi-Task	A very small training time for both RS and GS.
Bayes	The only case where R^2 was higher in GS.
Elastic Net	The higher divergence between training time in RS and GS.
Ridge	The smallest training time for GS.
LARS	Poor performance (R^2 and EVS) when GS hyperparameter tunning was used.
SGD	Worse MAE when RS was used.
SVM	Did not achieve acceptable accuracies in both tuning methods.
ELM	Much lesser training time than the equally complex algorithms, such as SVM and MLP. And better precision accuracies than MLP and SVM.
KNN	The third best MAE in RS and an acceptable training time**.**

Source: Elaborate by the authors

Conclusions

For both RS and GS, the Multi-layer Perceptron (MLP) returned the best accuracy measured by MAE, although at a very high training time. The hyperparameter tuned by GS took more than 15hours. At a much lower training time, the Multi-task Elastic-Net (Multi-Task) gave MAE comparable to MLP, and even a better R2 and EVS, for both RS and GS.

The Support Vector Machine algorithm (SVM) did not achieve the best performance in both tuning methods. The results suggest that while using the SVM algorithm for regression, the RS choice should be privileged over GS.

Multi-layer Perceptron (MLP), Support Vector Machine (SVM), and Extreme Learning Machines (ELM) algorithm have similar complexity, by significantly differ regarding the training time. The ELM gave better precision accuracies in a much lesser training time than MLP and

SVM. Overall, the results suggest that is better to use RS hyperparameter tuning for any of the algorithm's choices[3].

References

Bergstra, J. and Bengio, Y. Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research. v. 13 (10), p. 281−305, 2012. Available in: https://www.jmlr.org/papers/v13/bergstra12a.html

Bergstra, J., Bardenet, R., Yoshua, B. and Kégl, B. Algorithms for Hyper-Parameter Optimization. 25th Annual Conference on Neural Information Processing Systems (NIPS 2011), Dec 2011, Granada, Spain.

Brochu, E., Vlad, C. and Freitas. N. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning, CoRR, 2010.

Feurer, M. and Hutter, F. Hyperparameter Optimization. Chapter 1 in "Automated Machine Learning: Methods, Systems, Challenges", Org: Hutter, F., Kotthoff, L. and Vanschoren, J. The Springer Series on Challenges in Machine Learning, 2019.

Hastie, T. Tibshirani, R. Friedman, J. The Elements of Statistical Learning. Data Mining, Inference and Prediction. Springer Series in Statistics. 2Ed.

Snoek, J., Larochelle, H. and Adams, R. Practical Bayesian Optimization of Machine Learning Algorithms. Advances in Neural Information Processing Systems. v. 25, p. 2960-2968, 2012. Available in: https://arxiv.org/abs/1206.2944

[3] The updated database for years 2018 and 2019 will be available in the repository: https://github.com/estatistica/dados.

Appendix A: Optimal Parameters Chosen by Random Search

Mod	Best Hyperparameters
MLP	{'warm_start': True, 'tol': 8.02e-06, 'solver': 'lbfgs', 'power_t': 2.02, 'nesterovs_momentum': True, 'momentum': 0.055, 'learning_rate_init': 0.001, 'learning_rate': 'constant', 'hidden_layer_sizes': (200, 25), 'epsilon': 0.035, 'early_stopping': False, 'beta_2': 0.065, 'beta_1': 0.005, 'alpha': 0.005, 'activation': 'relu'}
Lasso	{'tol': 0.00233, 'selection': 'cyclic', 'precompute': True, 'positive': False, 'normalize': False, 'fit_intercept': False, 'copy_X': True, 'alpha': 0.01}
Multi-Task	{'warm_start': False, 'tol': 0.0014, 'selection': 'cyclic', 'normalize': False, 'l1_ratio': 4.366666666666666, 'fit_intercept': False, 'copy_X': False, 'alpha': 0.005}
Bayes	{'tol': 0.0192, 'normalize': False, 'lambda_2': 6.275e-06, 'lambda_1': 4.1275e-05, 'fit_intercept': False, 'copy_X': False, 'compute_score': True, 'alpha_2': 4.5025e-05, 'alpha_1': 1.275e-06}
Elastic Net	{'warm_start': False, 'tol': 0.131, 'selection': 'cyclic', 'precompute': True, 'positive': True, 'normalize': False, 'l1_ratio': 3.46, 'fit_intercept': False, 'copy_X': True, 'alpha': 0.012}
Ridge	{'tol': 0.00681, 'solver': 'cholesky', 'normalize': False, 'fit_intercept': True, 'copy_X': False, 'alpha': 0.036}
LARS	{'positive': False, 'normalize': True, 'fit_path': False, 'fit_intercept': False, 'eps': 22.0, 'copy_X': False, 'alpha': 0.03667}
SGD	{'tol': 0.0012, 'power_t': 0.4833, 'penalty': 'l2', 'loss': 'epsilon_insensitive', 'learning_rate': 'adaptive', 'l1_ratio': 0.2, 'eta0': 0.12583, 'epsilon': 0.0143, 'early_stopping': False, 'alpha': 3.8}
SVM	{'tol': 0.028, 'shrinking': True, 'kernel': 'sigmoid', 'gamma': 'auto', 'epsilon': 0.2, 'degree': 0.15, 'coef0': 0.68, 'C': 2.8}
ELM	{'hidden_layer__rbf_width': 0.0125, 'hidden_layer__n_hidden': 18, 'hidden_layer__activation_func': 'multiquadric', 'hidden_layer__activation_args': {'power': 3.5}}
KNN	{'weights': 'distance', 'p': 3, 'n_neighbors': 8, 'leaf_size': 30, 'algorithm': 'auto'}

Appendix B: Optimal Parameters Chosen by Grid Search

Mod	Best Hyperparameters
MLP	{'activation': 'relu', 'alpha': 0.105, 'beta_1': 0.005, 'beta_2': 0.055, 'early_stopping': False, 'epsilon': 0.005, 'hidden_layer_sizes': (75, 50, 75), 'learning_rate': 'constant', 'learning_rate_init': 0.016, 'momentum': 0.005, 'nesterovs_momentum': True, 'power_t': 3.02, 'solver': 'lbfgs', 'tol': 2e-08, 'warm_start': False}
Lasso	{'alpha': 0.015, 'copy_X': True, 'fit_intercept': True, 'normalize': False, 'positive': False, 'precompute': False, 'selection': 'random', 'tol': 0.0066}
Multi-Task	{'alpha': 0.005, 'copy_X': True, 'fit_intercept': True, 'l1_ratio': 0.03333333333333333, 'normalize': False, 'selection': 'cyclic', 'tol': 0.0002, 'warm_start': False}
Bayes	{'alpha_1': 2.5e-08, 'alpha_2': 4.7525e-05, 'compute_score': False, 'copy_X': True, 'fit_intercept': True, 'lambda_1': 4.7525e-05, 'lambda_2': 4.7525e-05, 'normalize': False, 'tol': 0.0102}
Elastic-Net	{'alpha': 0.0033333333333333335, 'copy_X': True, 'eps': 10.0, 'fit_intercept': True, 'fit_path': True, 'normalize': True, 'positive': False}
Ridge	{'alpha': 0.002, 'copy_X': True, 'fit_intercept': True, 'l1_ratio': 0.02, 'normalize': False, 'positive': False, 'precompute': False, 'selection': 'cyclic', 'tol': 0.021, 'warm_start': False}
LARS	{'alpha': 0.0033333333333333335, 'copy_X': True, 'eps': 10.0, 'fit_intercept': True, 'fit_path': True, 'normalize': True, 'positive': False}
SGD	{'alpha': 0.8, 'early_stopping': False, 'epsilon': 0.37, 'eta0': 0.0008333333333333334, 'l1_ratio': 0.19, 'learning_rate': 'invscaling', 'loss': 'squared_loss', 'penalty': 'l2', 'power_t': 0.26666666666666666}
SVM	{'alpha': 0.8, 'early_stopping': False, 'epsilon': 0.37, 'eta0': 0.0008333333333333334, 'l1_ratio': 0.19, 'learning_rate': 'invscaling', 'loss': 'squared_loss', 'penalty': 'l2', 'power_t': 0.26666666666666666}
ELM	{'hidden_layer__activation_args': {'power': 4.0}, 'hidden_layer__activation_func': 'inv_multiquadric', 'hidden_layer__n_hidden': 14, 'hidden_layer__rbf_width': 0.0125}
KNN	{'weights': 'uniform', 'p': 2, 'n_neighbors': 8, 'leaf_size': 10, 'algorithm': 'auto'}